

---

**NATIONAL WEATHER SERVICE  
OFFICE of HYDROLOGIC DEVELOPMENT**

---

**General Software Development Standards and Guidelines**

**Science Infusion Software Engineering Process Group  
(SISEPG)**

## Revision History

Date	Version	Description	Author
10/13/2004	1.0	Initial Version	B. Lawrence
04/18/2006	2.0	Reformat	M. Andre
05/11/2006	3.0	Revision to standards	B. Jackson
05/16/2006	3.1		
05/23/2006	3.2	Edit and changes	M. Andre, B. Lawrence
06/09/2006	3.3	Additional edits	M. Andre
06/21/2006	3.4	Apply HSEB Chief recommendations	M. Andre

## **Table of Contents**

Revision History .....	i
Table of Contents .....	ii
1. Introduction .....	1
2. Internal Documentation Standards .....	2
3. Coding Standards .....	3
3.1 Indentation .....	3
3.2 Inline Comments .....	3
3.3 Structured Programming .....	4
3.4 Classes, Subroutines, Functions, and Methods .....	4
3.5 Source Files .....	4
3.6 Variable Names .....	4
3.7 Use of Braces .....	4
3.8 Warnings .....	5
4. Coding Guidelines .....	5
4.1 Line Length .....	6
4.2 Spacing .....	6
4.3 Wrapping Lines .....	6
4.4 Variable Declarations .....	7
4.5 Program Statements .....	7
4.6 Use of Parentheses .....	8
4.7 Inline Comments .....	9
4.8 Coding for Efficiency vs. Coding for Readability .....	9
4.9 Meaningful Error Messages .....	10
4.10 Reasonably Sized Functions and Methods .....	10
4.11 Number of Routines per File .....	10
Appendix A - Internal Documentation Templates .....	11
Appendix B - Documentation Example .....	12

## **1. Introduction**

Software development standards help to ensure that software is written in a structured and easy to understand fashion. Well structured and documented software is easier to maintain. It reduces the amount of time needed for a new developer to learn how a program works. It also aids in the debugging process.

Software maintenance continues for many years after the code has been initially written. The task of maintaining the code is often passed from developer to developer. If the developers do not adhere to a basic set of software coding rules, different programming styles will result in code which is hard to read, understand, and or maintain.

Computer software forms the corner stone of most OHD applications. The Office of Hydrologic Development (OHD) can benefit greatly from well-written computer software. Well structured and documented software requires less time is required to maintain. The time saved can be applied towards other development endeavors. In short, software development standards lead to improved productivity.

The Science Infusion Software Engineering Process Group (SISEPG) has identified general and language specific software standards. The general level contains basic standards and guidelines which apply regardless of the programming language being used. The language specific standards stem from the syntax and semantics of a particular language.

In performing this task, the SISEPG is being careful not to produce a set of rules which restrict programmer individuality and creativity. If the standards are too strict, they will not be followed. A distinction has been made between standards and guidelines. Standards are items which programmers are expected to do. Guidelines can be viewed as suggestions which can help programmers write better software. The use of standards will be enforced. Following guidelines is optional but highly recommended.

Standard enforcement will be performed through peer software reviews. The format and level of formality of these reviews still needs to be determined.

This document details the General Software Development Standards and Guidelines.

## **2. Internal Documentation Standards**

If done correctly, internal documentation improves the readability of a software module. Many of the general software development guidelines are focused on using good internal documentation practices. The SISPEG has agreed that a file containing one or more software modules or a shell script file should have a comment block at its beginning containing the following basic information:

- The name of the file
- The name of the author who created the file
- The date the file was created
- The author's development group (e.g. HSEB, HSMB)
- Description (overview of the purpose of the modules)
- The names of each module contained in this file

Note that a module is a method, function, or subroutine.

Each module contained within the source file should be preceded by a block of comments showing the following:

(Items in parentheses are optional)

- The name of the module
- The name of the original author
- The date the module was created
- A description of what the module does
- A list of the calling arguments, their types, and brief explanations of what they do
- A list of required files and/or database tables needed by the routine, indicating if the routine expects the database or files to be already opened
- (All of the non system routines called by this module)
- Return values
- Error codes/exceptions
- (Operating System (OS) specific assumptions, e.g. this routine expects binary files to be Little Endian or this routine uses OS specific language extensions.)
- (A list of locally defined variables, their types, and how they are used)
- Modification history indicating who made modifications, when the mods were made, and what was done.

Appendix A of this document contains internal documentation templates. Appendix B contains an example of software modules which use these internal documentation standards.

## **3. Coding Standards**

General coding standards pertain to how the developer writes code. The SISEPG has come up with a small set of items it feels should be followed regardless of the programming language being used.

### **3.1 Indentation**

Proper and consistent indentation is important in producing easy to read and maintainable programs. Indentation should be used to:

- Emphasize the body of a control statement such as a loop or a select statement
- Emphasize the body of a conditional statement
- Emphasize a new scope block

A minimum of 3 spaces shall be used to indent. Generally, indenting by three or four spaces is considered to be adequate. Once the programmer chooses the number of spaces to indent by, then it is important that this indentation amount be consistently applied throughout the program. **Tabs shall not be used for indentation purposes.**

Examples:

```
/* Indentation used in a loop construct. Four spaces are used for
indentation. */
for ( int i = 0 ; i < number_of_employees ; ++i )
{
    total_wages += employee [ i ] . wages ;
}

// Indentation used in the body of a method.
package void get_vehicle_info ( )
{
    System.out.println ( "VIN: " + vin ) ;
    System.out.println ( "Make: " + make ) ;
    System.out.println ( "Model: " + model ) ;
    System.out.println ( "Year: " + year ) ;
}

/* Indentation used in a conditional statement. */
    IF ( IOS .NE. 0 )
        WRITE ( * , 10 ) IOS
    ENDIF

10    FORMAT ( "Error opening log file: ", I4 )
```

### **3.2 Inline Comments**

Inline comments explaining the functioning of the subroutine or key aspects of the algorithm shall be frequently used. See section 4.0 for guidance on the usage of inline comments.

### **3.3 Structured Programming**

Structured (or modular) programming techniques shall be used. GO TO statements shall not be used as they lead to “spaghetti” code, which is hard to read and maintain, except as outlined in the FORTRAN Standards and Guidelines.

### **3.4 Classes, Subroutines, Functions, and Methods**

Keep subroutines, functions, and methods reasonably sized. This depends upon the language being used. For guidance on how large to make software modules and methods, see section 4.0. A good rule of thumb for module length is to constrain each module to one function or action (i.e. each module should only do one “thing”). If a module grows too large, it is usually because the programmer is trying to accomplish too many actions at one time.

The names of the classes, subroutines, functions, and methods shall have verbs in them. That is the names shall specify an action, e.g. “get\_name”, “compute\_temperature”.

### **3.5 Source Files**

The name of the source file or script shall represent its function. All of the routines in a file shall have a common purpose.

### **3.6 Variable Names**

Variable shall have mnemonic or meaningful names that convey to a casual observer, the intent of its use. Variables shall be initialized prior to its first use.

### **3.7 Use of Braces**

In some languages, braces are used to delimit the bodies of conditional statements, control constructs, and blocks of scope. Programmers shall use either of the following bracing styles:

```
for (int j = 0 ; j < max_iterations ; ++j)
{
    /* Some work is done here. */
}
```

or the Kernighan and Ritchie style:

```
for ( int j = 0 ; j < max_iterations ; ++j ) {
    /* Some work is done here. */
}
```

It is felt that the former brace style is more readable and leads to neater-looking code than the latter style, but either use is acceptable. Whichever style is used, be sure to be consistent throughout the code. When editing code written by another author, adopt the style of bracing used.

Braces shall be used even when there is only one statement in the control block. For example:

Bad:

```
if (j == 0)
    printf ("j is zero.\n");
```

Better:

```
if (j == 0)
{
    printf ("j is zero.\n");
}
```

### **3.8 Compiler Warnings**

Compilers often issue two types of messages: warnings and errors. Compiler warnings normally do not stop the compilation process. However, compiler errors do stop the compilation process, forcing the developer to fix the problem and recompile.

Compiler and linker warnings shall be treated as errors and fixed. Even though the program will continue to compile in the presence of warnings, they often indicate problems which may affect the behavior, reliability and portability of the code.

Some compilers have options to suppress or enhance compile-time warning messages. Developers shall study the documentation and/or man pages associated with a compiler and choose the options which fully enable the compiler's code-checking features.

For example the `-Wall` option fully enables the gcc code checking features and should always be used:

```
gcc -Wall
```

## **4. Coding Guidelines**

General coding guidelines provide the programmer with a set of best practices which can be used to make programs easier to read and maintain. Unlike the coding standards, the use of these guidelines is not mandatory. However, the programmer is encouraged to review them and attempt to incorporate them into his/her programming style where appropriate.

Most of the examples use the C language syntax but the guidelines can be applied to all languages.



## **4.1 Line Length**

It is considered good practice to keep the lengths of source code lines at or below 80 characters. Lines longer than this may not be displayed properly on some terminals and tools. Some printers will truncate lines longer than 80 columns. FORTRAN is an exception to this standard. Its line lengths cannot exceed 72 columns.

## **4.2 Spacing**

The proper use of spaces within a line of code can enhance readability. Good rules of thumb are as follows:

- A keyword followed by a parenthesis should be separated by a space.
- A blank space should appear after each comma in an argument list.
- All binary operators except “.” should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment (“++”), and decrement(“--”) from their operands.
- Casts should be made followed by a blank space.

Example:

Bad:

```
cost=price+(price*sales_tax);  
fprintf(stdout,"The total cost is %5.2f\n",cost);
```

Better:

```
cost = price + ( price * sales_tax );  
fprintf ( stdout, "The total cost is %5.2f\n", cost) ;
```

## **4.3 Wrapping Lines**

When an expression will not fit on a single line, break it according to these following principles:

- Break after a comma

**Example:**

```
fprintf ( stdout , "\nThere are %d reasons to use standards\n" ,  
          num_reasons ) ;
```

- Break after an operator

**Example:**

```
long int total_apples = num_my_apples + num_his_apples +  
                        num_her_apples ;
```

- Prefer higher-level breaks to lower-level breaks

**Example:**

Bad:

```
longName1 = longName2 * (longName3 + LongName4 -  
                        longName5) + 4 * longName6 ;
```

Better:

```
longName1 = longName2 * (longName3 + LongName4 - LongName5)  
                + 4 * longName6 ;
```

- Align the new line with the beginning of the expression at the same level on the previous line.

**Example:**

```
total_windows = number_attic_windows + number_second_floor_windows  +  
                number_first_floor_windows ;
```

## **4.4 Variable Declarations**

Variable declarations that span multiple lines should always be preceded by a type.

**Example:**

Acceptable:

```
int price , score ;
```

Acceptable:

```
int price ;  
int score ;
```

Not Acceptable:

```
int price ,  
    score ;
```

## **4.5 Program Statements**

Program statements should be limited to one per line. Also, nested statements should be avoided when possible.

**Example:**

Bad:

```
number_of_names = names.length ; b = new JButton [ number_of_names ] ;
```

Better:

```
number_of_names = names.length ;  
b = new JButton [ number_of_names ] ;
```

### **Example:**

Bad:

```
strncpy ( city_name , string , strlen ( string ) ) ;
```

Better:

```
length = strlen ( string ) ;  
strncpy ( city_name , string , length ) ;
```

### **Example:**

Bad:

```
if ( ! strcmp ( string1 , string2 ) )  
{  
    fprintf ( stdout , "The strings are equal!\n" ) ;  
}
```

Better:

```
status = strcmp ( string1 , string2 ) ;  
  
if ( status == 0 )  
{  
    fprintf ( stdout , "The strings are equal!\n" ) ;  
}
```

## **4.6 Use of Parentheses**

It is better to use parentheses liberally. Even in cases where operator precedence unambiguously dictates the order of evaluation of an expression, often it is beneficial from a readability point of view to include parentheses anyway.

### **Example:**

Acceptable:

```
total = 3 - 4 * 3 ;
```

Better:

```
total = 3 - ( 4 * 3 ) ;
```

Even better:

```
total = ( -4 * 3 ) + 3 ;
```

### **Example:**

Acceptable:

```
if ( a == b && c == d )
```

Better:

```
if ( ( a == b ) && ( c == d ) )
```

## **4.7 Inline Comments**

Inline comments promote program readability. They allow a person not familiar with the code to more quickly understand it. It also helps the programmer who wrote the code to remember details forgotten over time. This reduces the amount of time required to perform software maintenance tasks.

As the name suggests, inline comments appear in the body of the source code itself. They explain the logic or parts of the algorithm which are not readily apparent from the code itself. Inline comments can also be used to describe the task being performed by a block of code.

Inline comments should be used to make the code clearer to a programmer trying to read and understand it. Writing a well structured program lends much to its readability even without inline comments. The bottom line is to use inline comments where they are needed to explain complicated program behavior or requirements. Use inline comments to generalize what a block of code, conditional structure, or control structure is doing. Do not use overuse inline comments to explain program details which are readily obvious to an intermediately skilled programmer.

A rule of thumb is that inline comments should make up 20% of the total lines of code in a program, excluding the header documentation blocks.

## **4.8 Coding for Efficiency vs. Coding for Readability**

There are many aspects to programming. These include writing software that runs efficiently and writing software that is easy to maintain. These two goals often collide with each other. Creating code that runs as efficiently as possible often means writing code that uses tricky logic and complex algorithms, code that can be hard to follow and maintain even with ample inline comments.

The programmer needs to carefully weigh efficiency gains versus program complexity and readability. If a more complicated algorithm offers only small gains in the speed of a program, the programmer should consider using a simpler algorithm. Although slower, the simpler algorithm will be easier for other programmers to understand.

## **4.9    *Meaningful Error Messages***

Error handling is an important aspect of computer programming. This not only includes adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

The required time should be spent writing code to detect errors. C and FORTRAN place the burden of the error handling on the programmer. Object oriented languages such as C++ and Java offer exception handling utilities for handling errors.

Error messages should be meaningful. When possible, they should indicate what the problem is, where the problem occurred, and when the problem occurred. A useful Java exception handling feature is the option to show a stack trace, which shows the sequence of method calls which led up to the exception.

Code which attempts to acquire system resources such as dynamic memory or files should always be tested for failure.

Error messages should be stored in way that makes them easy to review. For non interactive applications, such as a program which runs as part of a cron job, error messages should be logged into a log file. Interactive applications can either send error messages to a log file, standard output, or standard error. Interactive applications can also use popup windows to display error messages.

## **4.10   *Reasonably Sized Functions and Methods***

Software modules and methods should not contain an excessively large number of lines of code. They should be written to perform one specific task. If they become too long, then chances are the task being performed can be broken down into subtasks which can be handled by new routines or methods.

A reasonable number of lines of code for routine or a method is 200. This does not include documentation, either in the function/method header or inline comments.

## **4.11   *Number of Routines per File***

It is much easier to sort through code you did not write and you have never seen before if there are a minimal number of routines per file. This is only applicable to procedural languages such as C and FORTRAN. It does not apply to C++ and Java where there tends to be one public class definition per file.

## Appendix A - Internal Documentation Templates

Template internal documentation blocks. The examples use a C comment style which is also applicable to C++ and Java. When documenting other languages, use the comment style applicable to that language.

### File Documentation Block

**Definition:** This documentation block is located at the beginning of the file.

```
/******  
* Filename:  
* Original Author:  
* File Creation Date:  
* Development Group:  
* Description:  
* Modules:  
*     Name of Module 1  
*     Name of Module 2  
*     Name of Module 3  
*     Name of Module N  
*  
*****/
```

### Module Documentation Block

**Definition:** This documentation block precedes each module in the source file.

(Optional items are denoted through the use of parentheses.)

```
/******  
* Module Name:  
* Original Author:  
* Module Creation Date:  
* Description:  
* Calling Arguments:  
* Name      Input/Output      Type      Description  
* Required Files/Databases:  
* (Non System Routines Called:  
*   Name      Description)  
* Return Value:  
* Type      Description  
* Error Codes/Exceptions:  
* (OS Specific Assumptions if any):  
* (Local Variables:  
*   Type      Name      Description)  
* (Global Variables Used:  
*   Type      Name      Origin      Description)  
* (Constant and Macro Substitutions:  
*   Name      Header File      Description)  
* Modification History:  
*   Date      Developer      Action  
*****/
```

## Appendix B - Documentation Example

```
/*
 * Filename: convert_temperature.c
 *
 * Original Author: Jane Doe
 *
 * File Creation Date: August 20, 2004
 *
 * Development Group: OHD / HSEB
 *
 * Description: Contains routines for converting temperatures from
 *              Celsius to Fahrenheit.
 *
 * Modules:
 *   convert_C_to_F
 *   convert_F_to_C
 */
*****

/* Include files, definitions, globals go here. */
#include <convert_temperature.h>

#define ABSOLUTE_ZERO_IN_C      -273.15

/*
 * Module Name: convert_C_to_F
 *
 * Original Author: Jane Doe
 *
 * Module Creation Date: August 20, 2004
 *
 * Description: This function converts a Celsius temperature to Fahrenheit
 *
 * Calling Arguments:
 *
 *   Name           Input/Output   Type      Description
 *   temperatureC   Input          float      The Celsius temperature to be
 *                                           converted.
 *   temperatureF   Output         float*     The Celsius temperature
 *                                           converted to Fahrenheit.
 *
 * Required Files/Databases:
 *   None
 *
 * Non System Routines Called:
 *   None
 *
 * Return Value:
 *
 *   Type           Description
 *   int            The status of the conversion. Did the conversion
 *                   succeed or fail?
 *
 * Error Codes/Exceptions:
 *   CONVERSION_OK   The conversion succeeded.
 *   BELOW_ABSOLUTE_ZERO The supplied Celsius temperature is below
 *                       absolute zero. (These are defined in

```

**NOAA National Weather Service NWS/OHD**  
**General Software Coding Standards and Guidelines**

```
*                                     convert_temperature.h)
*
* OS Specific Assumptions:
*   None
*
* Local Variables:
*   Name      Type      Description
*   status    int       Contains the return status of this routine.
*
* Modification History:
*   Date      Developer  Action
*   8/20/2004  Jane Doe   Changed the equation used to convert
*                       Celsius to Fahrenheit.
*
*****/
int convert_C_to_F ( float temperatureC , float *temperatureF )
{
    int status = CONVERSION_OK ;

    /* Check to make sure the supplied temperature is at or above
       absolute zero. */
    if ( temperatureC < ABSOLUTE_ZERO_IN_C )
    {
        /* The user-supplied temperature in Celsius is below absolute
           zero.*/
        status = BELOW_ABSOLUTE_ZERO ;
    }
    else
    {
        /* Convert user-supplied temperature in Celsius to Fahrenheit. */
        *temperatureF = ( 9 * temperatureC / 5 ) + 32 ;
    }

    return status ;
}

/*****
* Module Name: convert_F_to_C
*
* Original Author: Jane Doe
*
* Module Creation Date: September 15, 2004
*
* Description: This function converts a Fahrenheit temperature to a
*              Celsius temperature.
*
* Calling Arguments:
*   Name      Input/Output  Type      Description
*   temperatureF  Input      float      The Fahrenheit temperature
*                                       to convert to Celsius.
*   temperatureC  Output      float*     The Celsius temperature
*                                       resulting from the
*                                       conversion.
*
* Required Files/Databases:
*   None

```



**NOAA National Weather Service NWS/OHD**  
**General Software Coding Standards and Guidelines**

```
*
* Non System Routines Called:
*   None
*
* Return Value:
*   Type           Description
*   int            Indicates whether the conversion was successful or
*                   failed.
*
* Error Codes/Exceptions:
*   CONVERSION_OK   - The conversion from Fahrenheit to Celsius
*                   was successful.
*
*   BELOW_ABSOLUTE_ZERO - The conversion from Fahrenheit to Celsius
*                   resulted in a temperature below absolute
*                   zero. (These values are defined in
*                   in convert_temperature.h)
*
* OS Specific Assumptions:
*   This routine can run on HP-UX or Linux.
*
* Local Variables:
*   Name      Type      Description
*   status    int       Contains the return status of this conversion
*                   function.
*
* Modification History:
*   Date      Developer      Action
*
*****/
int convert_F_to_C ( float temperatureF , float *temperatureC )
{
    int status = CONVERSION_OK;

    *temperatureC = ( temperatureF - 32 ) * 5 / 9;

    if ( *temperatureC < ABSOLUTE_ZERO_IN_C )
    {
        status = BELOW_ABSOLUTE_ZERO;
    }

    return status;
}
```